Probabilistic Hierarchy

Johannes Hölzl Andreas Lochbihler Dmitriy Traytel

February 23, 2021

Contents

1 Bisimilarity

definition *bisimilar* **where**
bisimilar Q s1 s2 x y ≡ (∃ *R*. *R x y* ∧ (∀ *x y*. *R x y* ⟶ *Q R* (*s1 x*) (*s2 y*)))

abbreviation *bisimilar-mc* ≡ *bisimilar* (λ*R*. *rel-pmf R*)
abbreviation *bisimilar-dlts* ≡ *bisimilar* (λ*R*. *rel-fun* (=) (*rel-option R*))
abbreviation *bisimilar-lts* ≡ *bisimilar* (λ*R*. *rel-bset* (*rel-prod* (=) *R*))
abbreviation *bisimilar-react* ≡ *bisimilar* (λ*R*. *rel-fun* (=) (*rel-option* (*rel-pmf R*)))
abbreviation *bisimilar-lmc* ≡ *bisimilar* (λ*R*. *rel-prod* (=) (*rel-pmf R*))
abbreviation *bisimilar-lmdp* ≡ *bisimilar* (λ*R*. *rel-prod* (=) (*rel-nebset* (*rel-pmf R*)))
abbreviation *bisimilar-gen* ≡ *bisimilar* (λ*R*. *rel-option* (*rel-pmf* (*rel-prod* (=) *R*)))
abbreviation *bisimilar-str* ≡ *bisimilar* (λ*R*. *rel-sum* (*rel-pmf R*) (*rel-option* (*rel-prod* (=) *R*)))

abbreviation *bisimilar-alt* ≡ *bisimilar* (λ*R*. *rel-sum* (*rel-pmf R*) (*rel-bset* (*rel-prod* (=) *R*)))
abbreviation *bisimilar-sseg* ≡ *bisimilar* (λ*R*. *rel-bset* (*rel-prod* (=) (*rel-pmf R*)))
abbreviation *bisimilar-seg* ≡ *bisimilar* (λ*R*. *rel-bset* (*rel-pmf* (*rel-prod* (=) *R*)))
abbreviation *bisimilar-bun* ≡ *bisimilar* (λ*R*. *rel-pmf* (*rel-bset* (*rel-prod* (=) *R*)))
abbreviation *bisimilar-pz* ≡ *bisimilar* (λ*R*. *rel-bset* (*rel-pmf* (*rel-bset* (*rel-prod* (=) *R*))))
abbreviation *bisimilar-mg* ≡ *bisimilar* (λ*R*. *rel-bset* (*rel-pmf* (*rel-bset* (*rel-sum* (*rel-prod* (=) *R*) *R*))))

2 Systems

codatatype *mc* = *MC mc pmf*
codatatype *'a dlts* = *DLTS 'a* ⇒ *'a dlts option*
codatatype (*'a*, *'k*) *lts* = *LTS* (*'a* × (*'a*, *'k*) *lts*) *set*[*'k*]
codatatype *'a react* = *React 'a* ⇒ *'a react pmf option*
codatatype *'a lmc* = *LMC 'a* × *'a lmc pmf*
codatatype (*'a*, *'k*) *lmdp* = *LMDP 'a* × (*'a*, *'k*) *lmdp pmf set*![*'k*]
codatatype *'a gen* = *Gen* (*'a* × *'a gen*) *pmf option*
codatatype *'a str* = *Str 'a str pmf* + (*'a* × *'a str*) *option*
codatatype (*'a*, *'k*) *alt* = *Alt* (*'a*, *'k*) *alt pmf* + (*'a* × (*'a*, *'k*) *alt*) *set*[*'k*]
codatatype (*'a*, *'k*) *sseg* = *SSeg* (*'a* × (*'a*, *'k*) *sseg pmf*) *set*[*'k*]
codatatype (*'a*, *'k*) *seg* = *Seg* (*'a* × (*'a*, *'k*) *seg*) *pmf set*[*'k*]
codatatype (*'a*, *'k*) *bun* = *Bun* ((*'a* × (*'a*, *'k*) *bun*) *set*[*'k*]) *pmf*
codatatype (*'a*, *'k1*, *'k2*) *pz* = *PZ* ((*'a* × (*'a*, *'k1*, *'k2*) *pz*) *set*[*'k1*]) *pmf set*[*'k2*]
codatatype (*'a*, *'k1*, *'k2*) *mg* = *MG* ((*'a* × (*'a*, *'k1*, *'k2*) *mg* + (*'a*, *'k1*, *'k2*) *mg*) *set*[*'k1*]) *pmf set*[*'k2*]

3 Unfolds

primcorec *unfold-mc* :: (*'a* ⇒ *'a pmf*) ⇒ *'a* ⇒ *mc* **where**
  unfold-mc s x = *MC* (*map-pmf* (*unfold-mc s*) (*s x*))

primcorec *unfold-dlts* :: (*'a* ⇒ *'b* ⇒ *'a option*) ⇒ *'a* ⇒ *'b dlts* **where**
  unfold-dlts s x = *DLTS* (*map-option* (*unfold-dlts s*) *o s x*)

primcorec *unfold-lts* :: (*'a* ⇒ (*'b* × *'a*) *set*[*'k*]) ⇒ *'a* ⇒ (*'b*, *'k*) *lts* **where**
  unfold-lts s x = *LTS* (*map-bset* (*map-prod id* (*unfold-lts s*)) (*s x*))

primcorec *unfold-react* :: (*'a* ⇒ *'b* ⇒ *'a pmf option*) ⇒ *'a* ⇒ *'b react* **where**
  unfold-react s x = *React* (*map-option* (*map-pmf* (*unfold-react s*)) *o s x*)

primcorec *unfold-lmc* :: (*'a* ⇒ *'b* × *'a pmf*) ⇒ *'a* ⇒ *'b lmc* **where**
  unfold-lmc s x = *LMC* (*map-prod id* (*map-pmf* (*unfold-lmc s*)) (*s x*))

primcorec *unfold-lmdp* :: (*'a* ⇒ *'b* × *'a pmf set*![*'k*]) ⇒ *'a* ⇒ (*'b*, *'k*) *lmdp* **where**
  unfold-lmdp s x = *LMDP* (*map-prod id* (*map-nebset* (*map-pmf* (*unfold-lmdp s*))) (*s x*))

primcorec *unfold-gen* :: (*'a* ⇒ ((*'b* × *'a*) *pmf*) *option*) ⇒ *'a* ⇒ *'b gen* **where**
unfold-gen s x = *Gen* (*map-option* (*map-pmf* (*map-prod id* (*unfold-gen s*))) (*s x*))

primcorec *unfold-str* :: (*'a* ⇒ *'a pmf* + (*'b* × *'a*) *option*) ⇒ *'a* ⇒ *'b str* **where**
unfold-str s x = *Str* (*map-sum* (*map-pmf* (*unfold-str s*)) (*map-option* (*map-prod id* (*unfold-str s*))) (*s x*))

primcorec *unfold-alt* :: (*'a* ⇒ *'a pmf* + (*'b* × *'a*) *set*[*'k*]) ⇒ *'a* ⇒ (*'b*, *'k*) *alt* **where**
unfold-alt s x = *Alt* (*map-sum* (*map-pmf* (*unfold-alt s*)) (*map-bset* (*map-prod id* (*unfold-alt s*))) (*s x*))

primcorec *unfold-sseg* :: (*'a* ⇒ (*'b* × *'a pmf*) *set*[*'k*]) ⇒ *'a* ⇒ (*'b*, *'k*) *sseg* **where**
unfold-sseg s x = *SSeg* (*map-bset* (*map-prod id* (*map-pmf* (*unfold-sseg s*))) (*s x*))

primcorec *unfold-seg* :: (*'a* ⇒ ((*'b* × *'a*) *pmf*) *set*[*'k*]) ⇒ *'a* ⇒ (*'b*, *'k*) *seg* **where**
unfold-seg s x = *Seg* (*map-bset* (*map-pmf* (*map-prod id* (*unfold-seg s*))) (*s x*))

primcorec *unfold-bun* :: (*'a* ⇒ ((*'b* × *'a*) *set*[*'k*]) *pmf*) ⇒ *'a* ⇒ (*'b*, *'k*) *bun* **where**
unfold-bun s x = *Bun* (*map-pmf* (*map-bset* (*map-prod id* (*unfold-bun s*))) (*s x*))

primcorec *unfold-pz* :: (*'a* ⇒ ((*'b* × *'a*) *set*[*'k1*]) *pmf set*[*'k2*]) ⇒ *'a* ⇒ (*'b*, *'k1*, *'k2*) *pz* **where**
unfold-pz s x = *PZ* (*map-bset* (*map-pmf* (*map-bset* (*map-prod id* (*unfold-pz s*)))) (*s x*))

primcorec *unfold-mg* :: (*'a* ⇒ ((*'b* × *'a* + *'a*) *set*[*'k1*]) *pmf set*[*'k2*]) ⇒ *'a* ⇒ (*'b*, *'k1*, *'k2*) *mg* **where**
unfold-mg s x = *MG* (*map-bset* (*map-pmf* (*map-bset* (*map-sum* (*map-prod id* (*unfold-mg s*)) (*unfold-mg s*)))) (*s x*))

4 Embeddings

abbreviation (*input*) *react-of-dlts-emb dlts* ≡ *map-option return-pmf o dlts*
abbreviation (*input*) *lts-of-dlts-emb* ≡ *bgraph*
abbreviation (*input*) *sseg-of-react-emb* ≡ *bgraph*
abbreviation (*input*) *gen-of-lmc-emb* ≡ *Some o case-prod* (*map-pmf o Pair*)
abbreviation (*input*) *lmdp-of-lmc-emb* ≡ *map-prod id nebsingleton*
abbreviation (*input*) *sseg-of-lmdp-emb* ≡ (λ(*a*, *X*). *map-bset* (*Pair a*) (*bset-of-nebset X*))
abbreviation (*input*) *sseg-of-lts-emb* ≡ *map-bset* (*map-prod id return-pmf*)
abbreviation (*input*) *ssegopt-of-alt-emb* ≡ *case-sum*
(*map-bset* (*Pair None*) *o bsingleton*)
(*map-bset* (*map-prod Some return-pmf*))
abbreviation (*input*) *bunopt-of-alt-emb* ≡ *case-sum*
(*map-pmf* (*bsingleton o Pair None*))
(*map-pmf* (*map-bset* (*map-prod Some id*)) *o return-pmf*)

abbreviation (*input*) *segopt-of-seg-emb* ≡ *map-bset* (*map-pmf* (*map-prod Some id*))
abbreviation (*input*) *ssegopt-of-sseg-emb* ≡ *map-bset* (*map-prod Some id*)
abbreviation (*input*) *bunopt-of-bun-emb* ≡ *map-pmf* (*map-bset* (*map-prod Some id*))
abbreviation (*input*) *pzopt-of-pz-emb* ≡ *map-bset* (*map-pmf* (*map-bset* (*map-prod Some id*)))
abbreviation (*input*) *seg-of-sseg-emb* ≡ *map-bset* (*case-prod* (*map-pmf o Pair*))
abbreviation (*input*) *pz-of-seg-emb* ≡ *map-bset* (*map-pmf bsingleton*)
abbreviation (*input*) *pz-of-bun-emb* ≡ *bsingleton*
abbreviation (*input*) *seg-of-gen-emb* ≡ *bset-of-option*
abbreviation (*input*) *bun-of-lts-emb* ≡ *return-pmf*
abbreviation (*input*) *bun-of-gen-emb* ≡ *case-option* (*return-pmf bempty*) (*map-pmf bsingleton*)
abbreviation (*input*) *str-of-mc-emb* ≡ *Inl*
abbreviation (*input*) *alt-of-str-emb* ≡ *map-sum id bset-of-option*
abbreviation (*input*) *pzopt-of-mg-emb* ≡ *map-bset* (*map-pmf* (*map-bset* (*case-sum* (*map-prod Some id*) (*Pair None*))))
abbreviation (*input*) *mg-of-pzopt-emb* ≡ *map-bset* (*map-pmf* (*map-bset* (λ(*a*, *s*). *case-option* (*Inr s*) (λ*a*. (*Inl* (*a*, *s*))) *a*)))

Obsolete edges (susumed by transitive ones)

abbreviation (*input*) *mg-of-pz-emb* ≡ *map-bset* (*map-pmf* (*map-bset Inl*))
abbreviation (*input*) *mg-of-alt1-emb* ≡ *case-sum*
(*map-bset* (*map-pmf* (*map-bset Inr o bsingleton*)) *o bsingleton*)
(*map-bset* (*map-pmf* (*map-bset Inl o bsingleton*) *o return-pmf*))
abbreviation (*input*) *mg-of-alt2-emb* ≡ *case-sum*
(*map-bset* (*map-pmf* (*map-bset Inr o bsingleton*)) *o bsingleton*)
(*map-bset* (*map-pmf* (*map-bset Inl*) *o return-pmf*) *o bsingleton*)
abbreviation (*input*) *pz-of-alt1-emb* ≡ *case-sum*
(*map-bset* (*map-pmf* (*map-bset* (*Pair None*) *o bsingleton*)) *o bsingleton*)
(*map-bset* (*map-pmf* (*map-bset* (*map-prod Some id*) *o bsingleton*) *o return-pmf*))
abbreviation (*input*) *pz-of-alt2-emb* ≡ *case-sum*
(*map-bset* (*map-pmf* (*map-bset* (*Pair None*) *o bsingleton*)) *o bsingleton*)
(*map-bset* (*map-pmf* (*map-bset* (*map-prod Some id*)) *o return-pmf*) *o bsingleton*)

definition *react-of-dlts* :: *'a dlts* ⇒ *'a react* **where**
[*simp*]: *react-of-dlts* = *unfold-react* (*react-of-dlts-emb o un-DLTS*)

definition *lts-of-dlts* :: *'a dlts* ⇒ (*'a*, *'a set*) *lts* **where**
[*simp*]: *lts-of-dlts* = *unfold-lts* (*lts-of-dlts-emb o un-DLTS*)

definition *sseg-of-react* :: *'a react* ⇒ (*'a*, *'a set*) *sseg* **where**
[*simp*]: *sseg-of-react* = *unfold-sseg* (*sseg-of-react-emb o un-React*)

definition *lmdp-of-lmc* :: *'a lmc* ⇒ (*'a*, *'k*) *lmdp* **where**
[*simp*]: *lmdp-of-lmc* = *unfold-lmdp* (*lmdp-of-lmc-emb o un-LMC*)

definition *gen-of-lmc* :: *'a lmc* ⇒ *'a gen* **where**
[*simp*]: *gen-of-lmc* = *unfold-gen* (*gen-of-lmc-emb o un-LMC*)

definition *sseg-of-lmdp* :: (*'a*, *'k*) *lmdp* ⇒ (*'a*, *'k*) *sseg* **where**
[*simp*]: *sseg-of-lmdp* = *unfold-sseg* (*sseg-of-lmdp-emb o un-LMDP*)

definition *sseg-of-lts* :: (*'a*, *'k*) *lts* ⇒ (*'a*, *'k*) *sseg* **where**
[*simp*]: *sseg-of-lts* = *unfold-sseg* (*sseg-of-lts-emb o un-LTS*)

definition *ssegopt-of-alt* :: (*'a*, *'k*) *alt* ⇒ (*'a option*, *'k*) *sseg* **where**
[*simp*]: *ssegopt-of-alt* = *unfold-sseg* (*ssegopt-of-alt-emb o un-Alt*)

definition *bunopt-of-alt* :: (*'a*, *'k*) *alt* ⇒ (*'a option*, *'k*) *bun* **where**
[*simp*]: *bunopt-of-alt* = *unfold-bun* (*bunopt-of-alt-emb o un-Alt*)

definition *seg-of-sseg* :: (*'a*, *'k*) *sseg* ⇒ (*'a*, *'k*) *seg* **where**
[*simp*]: *seg-of-sseg* = *unfold-seg* (*seg-of-sseg-emb o un-SSeg*)

definition *seg-of-gen* :: *'a gen* ⇒ (*'a*, *'k*) *seg* **where**
[*simp*]: *seg-of-gen* = *unfold-seg* (*seg-of-gen-emb o un-Gen*)

definition *bun-of-lts* :: (*'a*, *'k*) *lts* ⇒ (*'a*, *'k*) *bun* **where**
[*simp*]: *bun-of-lts* = *unfold-bun* (*bun-of-lts-emb o un-LTS*)

definition *bun-of-gen* :: *'a gen* ⇒ (*'a*, *'k*) *bun* **where**
[*simp*]: *bun-of-gen* = *unfold-bun* (*bun-of-gen-emb o un-Gen*)

definition *pz-of-seg* :: (*'a*, *'k*) *seg* ⇒ (*'a*, *'k1*, *'k*) *pz* **where**
[*simp*]: *pz-of-seg* = *unfold-pz* (*pz-of-seg-emb o un-Seg*)

definition *pz-of-bun* :: (*'a*, *'k*) *bun* ⇒ (*'a*, *'k*, *'k1*) *pz* **where**
[*simp*]: *pz-of-bun* = *unfold-pz* (*pz-of-bun-emb o un-Bun*)

definition *mg-of-pz* :: (*'a*, *'k1*, *'k2*) *pz* ⇒ (*'a*, *'k1*, *'k2*) *mg* **where**
[*simp*]: *mg-of-pz* = *unfold-mg* (*mg-of-pz-emb o un-PZ*)

definition *str-of-mc* :: *mc* ⇒ *'a str* **where**
[*simp*]: *str-of-mc* = *unfold-str* (*str-of-mc-emb o un-MC*)

definition *alt-of-str* :: *'a str* ⇒ (*'a*, *'k*) *alt* **where**
[*simp*]: *alt-of-str* = *unfold-alt* (*alt-of-str-emb o un-Str*)

definition *ssegopt-of-sseg* :: (*'a*, *'k*) *sseg* ⇒ (*'a option*, *'k*) *sseg* **where**
[*simp*]: *ssegopt-of-sseg* = *unfold-sseg* (*ssegopt-of-sseg-emb o un-SSeg*)

definition *segopt-of-seg* :: (*'a*, *'k*) *seg* ⇒ (*'a option*, *'k*) *seg* **where**
[*simp*]: *segopt-of-seg* = *unfold-seg* (*segopt-of-seg-emb o un-Seg*)

definition *bunopt-of-bun* :: (*'a*, *'k*) *bun* ⇒ (*'a option*, *'k*) *bun* **where**

[*simp*]: *bunopt-of-bun* = *unfold-bun* (*bunopt-of-bun-emb o un-Bun*)

definition *pzopt-of-pz* :: ('*a*, '*k1*, '*k2*) *pz* ⇒ ('*a option*, '*k1*, '*k2*) *pz* **where**
[*simp*]: *pzopt-of-pz* = *unfold-pz* (*pzopt-of-pz-emb o un-PZ*)

definition *pzopt-of-mg* :: ('*a*, '*k1*, '*k2*) *mg* ⇒ ('*a option*, '*k1*, '*k2*) *pz* **where**
[*simp*]: *pzopt-of-mg* = *unfold-pz* (*pzopt-of-mg-emb o un-MG*)

definition *mg-of-pzopt* :: ('*a option*, '*k1*, '*k2*) *pz* ⇒ ('*a*, '*k1*, '*k2*) *mg* **where**
[*simp*]: *mg-of-pzopt* = *unfold-mg* (*mg-of-pzopt-emb o un-PZ*)

definition *mg-of-alt1* :: ('*a*, '*k*) *alt* ⇒ ('*a*, '*k1*, '*k*) *mg* **where**
[*simp*]: *mg-of-alt1* = *unfold-mg* (*mg-of-alt1-emb o un-Alt*)

definition *mg-of-alt2* :: ('*a*, '*k*) *alt* ⇒ ('*a*, '*k*, '*k1*) *mg* **where**
[*simp*]: *mg-of-alt2* = *unfold-mg* (*mg-of-alt2-emb o un-Alt*)

definition *pz-of-alt1* :: ('*a*, '*k*) *alt* ⇒ ('*a option*, '*k1*, '*k*) *pz* **where**
[*simp*]: *pz-of-alt1* = *unfold-pz* (*pz-of-alt1-emb o un-Alt*)

definition *pz-of-alt2* :: ('*a*, '*k*) *alt* ⇒ ('*a option*, '*k*, '*k2*) *pz* **where**
[*simp*]: *pz-of-alt2* = *unfold-pz* (*pz-of-alt2-emb o un-Alt*)

5 Automation Setup

lemma *mc-rel-eq*[*unfolded vimage2p-def*]:
BNF-Def.vimage2p un-MC un-MC (*rel-pmf* (=)) = (=)
by (*auto simp add*: *vimage2p-def pmf.rel-eq option.rel-eq fun.rel-eq fun-eq-iff mc.expand*)

lemma *dlts-rel-eq*[*unfolded vimage2p-def*]:
BNF-Def.vimage2p un-DLTS un-DLTS (*rel-fun* (=) (*rel-option* (=))) = (=)
by (*auto simp add*: *vimage2p-def pmf.rel-eq option.rel-eq fun.rel-eq fun-eq-iff dlts.expand*)

lemma *react-rel-eq*[*unfolded vimage2p-def*]:
BNF-Def.vimage2p un-React un-React (*rel-fun* (=) (*rel-option* (*rel-pmf* (=)))) = (=)
by (*auto simp add*: *vimage2p-def pmf.rel-eq option.rel-eq fun.rel-eq fun-eq-iff react.expand*)

lemma *all-neq-Inl-ex-eq-Inr*[*dest*]: (∀ *l*. *x* ≠ *Inl l*) ⟹ (∃ *r*. *x* = *Inr r*) **by** (*cases x*) *auto*
lemma *all-neq-Inr-ex-eq-Inl*[*dest*]: (∀ *r*. *x* ≠ *Inr r*) ⟹ (∃ *l*. *x* = *Inl l*) **by** (*cases x*) *auto*
lemma *all2-neq-Inl-ex-eq-Inr*[*dest*]: (∀ *a b*. *x* ≠ *Inl* (*a*, *b*)) ⟹ (∃ *r*. *x* = *Inr r*) **by** (*cases x*) *auto*
lemma *all2-neq-Inr-ex-eq-Inl*[*dest*]: (∀ *a b*. *x* ≠ *Inr* (*a*, *b*)) ⟹ (∃ *l*. *x* = *Inl l*) **by** (*cases x*) *auto*

lemma *rel-prod-simp-asym*[*simp*]:
$\bigwedge x\ y.$ *rel-prod* $R\ S\ (x,\ y) = (\lambda z.$ *case* z *of* $(x',\ y') \Rightarrow R\ x\ x' \wedge S\ y\ y')$
$\bigwedge x\ y\ z.$ *rel-prod* $R\ S\ x\ (y,\ z) = ($*case* x *of* $(y',\ z') \Rightarrow R\ y'\ y \wedge S\ z'\ z)$
by *auto*

lemma *map-prod-eq-Pair-iff*[*simp*]:
map-prod $f\ g\ x = (y,\ z) \longleftrightarrow (f\ (fst\ x) = y \wedge g\ (snd\ x) = z)$
by (*cases* x) *auto*

lemmas [*abs-def*, *simp*] =
sum.rel-map prod.rel-map option.rel-map pmf.rel-map bset.rel-map fun.rel-map nebset.rel-map

lemmas [*simp*] =
lts.rel-eq lmc.rel-eq lmdp.rel-eq gen.rel-eq str.rel-eq alt.rel-eq sseg.rel-eq seg.rel-eq
bun.rel-eq pz.rel-eq mg.rel-eq
rel-pmf-return-pmf1 rel-pmf-return-pmf2 set-pmf-not-empty rel-pmf-rel-prod
bset.set-map nebset.set-map

lemmas [*simp del*] =
split-paired-Ex

lemma *bisimilar-eqI*:
assumes $\bigwedge R.\ [\![R\ x\ y;\ \bigwedge x\ y.\ R\ x\ y \Longrightarrow Q\ R\ (s1\ x)\ (s2\ y)]\!] \Longrightarrow\ P\ x\ y$
and $P\ x\ y \Longrightarrow \forall\, x\ y.\ P\ x\ y \longrightarrow Q\ P\ (s1\ x)\ (s2\ y)$
shows *bisimilar* Q *s1 s2* $x\ y = P\ x\ y$
using *assms* **unfolding** *bisimilar-def* **by** *auto*

bundle *probabilistic-hierarchy* =
rel-fun-def[*simp*]
sum.splits[*split*]
prod.splits[*split*]
option.splits[*split*]

predicate2-eqD[*THEN iffD2*, *OF mc-rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF dlts-rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF lts.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF react-rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF lmc.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF lmdp.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF gen.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF str.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF alt.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF sseg.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF seg.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF bun.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF pz.rel-eq*, *dest*]
predicate2-eqD[*THEN iffD2*, *OF mg.rel-eq*, *dest*]

iffD1[*OF lts.rel-sel*, *dest*!]
iffD1[*OF lmc.rel-sel*, *dest*!]
iffD1[*OF lmdp.rel-sel*, *dest*!]
iffD1[*OF gen.rel-sel*, *dest*!]
iffD1[*OF str.rel-sel*, *dest*!]
iffD1[*OF alt.rel-sel*, *dest*!]
iffD1[*OF sseg.rel-sel*, *dest*!]
iffD1[*OF seg.rel-sel*, *dest*!]
iffD1[*OF bun.rel-sel*, *dest*!]
iffD1[*OF pz.rel-sel*, *dest*!]
iffD1[*OF mg.rel-sel*, *dest*!]

pmf.rel-refl[*intro*]
bset.rel-refl[*intro*]
nebset.rel-refl[*intro*]
prod.rel-refl[*intro*]
sum.rel-refl[*intro*]
option.rel-refl[*intro*]

pmf.rel-mono-strong[*intro*]
bset.rel-mono-strong[*intro*]
nebset.rel-mono-strong[*intro*]
prod.rel-mono-strong[*intro*]
sum.rel-mono-strong[*intro*]
option.rel-mono-strong[*intro*]

6 Proofs

context
includes *probabilistic-hierarchy*
begin

method *bisimilar-alt* =
  rule bisimilar-eqI,
  match **conclusion in** *u1 s1 x* = *u2 s2 y* **for** *u1 u2 s1 s2 x y* ⇒
    ‹*coinduction arbitrary*: *x y*, *fastforce*›,
  fastforce

lemma *bisimilar-alt*:
  ⋀*s1 s2*. *bisimilar-mc s1 s2 x y* = (*unfold-mc s1 x* = *unfold-mc s2 y*)
  ⋀*s1 s2*. *bisimilar-dlts s1 s2 x y* = (*unfold-dlts s1 x* = *unfold-dlts s2 y*)
  ⋀*s1 s2*. *bisimilar-lts s1 s2 x y* = (*unfold-lts s1 x* = *unfold-lts s2 y*)
  ⋀*s1 s2*. *bisimilar-react s1 s2 x y* = (*unfold-react s1 x* = *unfold-react s2 y*)
  ⋀*s1 s2*. *bisimilar-lmc s1 s2 x y* = (*unfold-lmc s1 x* = *unfold-lmc s2 y*)
  ⋀*s1 s2*. *bisimilar-lmdp s1 s2 x y* = (*unfold-lmdp s1 x* = *unfold-lmdp s2 y*)
  ⋀*s1 s2*. *bisimilar-gen s1 s2 x y* = (*unfold-gen s1 x* = *unfold-gen s2 y*)
  ⋀*s1 s2*. *bisimilar-str s1 s2 x y* = (*unfold-str s1 x* = *unfold-str s2 y*)
  ⋀*s1 s2*. *bisimilar-alt s1 s2 x y* = (*unfold-alt s1 x* = *unfold-alt s2 y*)

$\bigwedge s1\ s2.\ bisimilar\text{-}sseg\ s1\ s2\ x\ y = (unfold\text{-}sseg\ s1\ x = unfold\text{-}sseg\ s2\ y)$
$\bigwedge s1\ s2.\ bisimilar\text{-}seg\ s1\ s2\ x\ y = (unfold\text{-}seg\ s1\ x = unfold\text{-}seg\ s2\ y)$
$\bigwedge s1\ s2.\ bisimilar\text{-}bun\ s1\ s2\ x\ y = (unfold\text{-}bun\ s1\ x = unfold\text{-}bun\ s2\ y)$
$\bigwedge s1\ s2.\ bisimilar\text{-}pz\ s1\ s2\ x\ y = (unfold\text{-}pz\ s1\ x = unfold\text{-}pz\ s2\ y)$
$\bigwedge s1\ s2.\ bisimilar\text{-}mg\ s1\ s2\ x\ y = (unfold\text{-}mg\ s1\ x = unfold\text{-}mg\ s2\ y)$
by *bisimilar-alt+*

method *commute-prover* =
intro ext,
match **conclusion in** $u1\ s1\ x = (emb\ o\ u2\ s2)\ x$ **for** $emb\ u1\ u2\ s1\ s2\ x \Rightarrow$
⟨*coinduction arbitrary*: *x*, *fastforce*⟩

lemma *emb-commute*:
$\bigwedge s.\ unfold\text{-}lts\ (lts\text{-}of\text{-}dlts\text{-}emb\ o\ s) = lts\text{-}of\text{-}dlts\ o\ unfold\text{-}dlts\ s$
$\bigwedge s.\ unfold\text{-}gen\ (gen\text{-}of\text{-}lmc\text{-}emb\ o\ s) = gen\text{-}of\text{-}lmc\ o\ unfold\text{-}lmc\ s$
$\bigwedge s.\ unfold\text{-}lmdp\ (lmdp\text{-}of\text{-}lmc\text{-}emb\ o\ s) = lmdp\text{-}of\text{-}lmc\ o\ unfold\text{-}lmc\ s$
$\bigwedge s.\ unfold\text{-}react\ (react\text{-}of\text{-}dlts\text{-}emb\ o\ s) = react\text{-}of\text{-}dlts\ o\ unfold\text{-}dlts\ s$
$\bigwedge s.\ unfold\text{-}sseg\ (sseg\text{-}of\text{-}lmdp\text{-}emb\ o\ s) = sseg\text{-}of\text{-}lmdp\ o\ unfold\text{-}lmdp\ s$
$\bigwedge s.\ unfold\text{-}sseg\ (sseg\text{-}of\text{-}lts\text{-}emb\ o\ s) = sseg\text{-}of\text{-}lts\ o\ unfold\text{-}lts\ s$
$\bigwedge s.\ unfold\text{-}sseg\ (ssegopt\text{-}of\text{-}alt\text{-}emb\ o\ s) = ssegopt\text{-}of\text{-}alt\ o\ unfold\text{-}alt\ s$
$\bigwedge s.\ unfold\text{-}sseg\ (sseg\text{-}of\text{-}react\text{-}emb\ o\ s) = sseg\text{-}of\text{-}react\ o\ unfold\text{-}react\ s$
$\bigwedge s.\ unfold\text{-}seg\ (seg\text{-}of\text{-}sseg\text{-}emb\ o\ s) = seg\text{-}of\text{-}sseg\ o\ unfold\text{-}sseg\ s$
$\bigwedge s.\ unfold\text{-}seg\ (seg\text{-}of\text{-}gen\text{-}emb\ o\ s) = seg\text{-}of\text{-}gen\ o\ unfold\text{-}gen\ s$
$\bigwedge s.\ unfold\text{-}bun\ (bun\text{-}of\text{-}lts\text{-}emb\ o\ s) = bun\text{-}of\text{-}lts\ o\ unfold\text{-}lts\ s$
$\bigwedge s.\ unfold\text{-}bun\ (bunopt\text{-}of\text{-}alt\text{-}emb\ o\ s) = bunopt\text{-}of\text{-}alt\ o\ unfold\text{-}alt\ s$
$\bigwedge s.\ unfold\text{-}bun\ (bun\text{-}of\text{-}gen\text{-}emb\ o\ s) = bun\text{-}of\text{-}gen\ o\ unfold\text{-}gen\ s$
$\bigwedge s.\ unfold\text{-}pz\ (pz\text{-}of\text{-}seg\text{-}emb\ o\ s) = pz\text{-}of\text{-}seg\ o\ unfold\text{-}seg\ s$
$\bigwedge s.\ unfold\text{-}pz\ (pz\text{-}of\text{-}bun\text{-}emb\ o\ s) = pz\text{-}of\text{-}bun\ o\ unfold\text{-}bun\ s$
$\bigwedge s.\ unfold\text{-}str\ (str\text{-}of\text{-}mc\text{-}emb\ o\ s) = str\text{-}of\text{-}mc\ o\ unfold\text{-}mc\ s$
$\bigwedge s.\ unfold\text{-}alt\ (alt\text{-}of\text{-}str\text{-}emb\ o\ s) = alt\text{-}of\text{-}str\ o\ unfold\text{-}str\ s$
$\bigwedge s.\ unfold\text{-}sseg\ (ssegopt\text{-}of\text{-}sseg\text{-}emb\ o\ s) = ssegopt\text{-}of\text{-}sseg\ o\ unfold\text{-}sseg\ s$
$\bigwedge s.\ unfold\text{-}seg\ (segopt\text{-}of\text{-}seg\text{-}emb\ o\ s) = segopt\text{-}of\text{-}seg\ o\ unfold\text{-}seg\ s$
$\bigwedge s.\ unfold\text{-}bun\ (bunopt\text{-}of\text{-}bun\text{-}emb\ o\ s) = bunopt\text{-}of\text{-}bun\ o\ unfold\text{-}bun\ s$
$\bigwedge s.\ unfold\text{-}pz\ (pzopt\text{-}of\text{-}pz\text{-}emb\ o\ s) = pzopt\text{-}of\text{-}pz\ o\ unfold\text{-}pz\ s$
$\bigwedge s.\ unfold\text{-}pz\ (pzopt\text{-}of\text{-}mg\text{-}emb\ o\ s) = pzopt\text{-}of\text{-}mg\ o\ unfold\text{-}mg\ s$
$\bigwedge s.\ unfold\text{-}mg\ (mg\text{-}of\text{-}pzopt\text{-}emb\ o\ s) = mg\text{-}of\text{-}pzopt\ o\ unfold\text{-}pz\ s$

$\bigwedge s.\ unfold\text{-}mg\ (mg\text{-}of\text{-}pz\text{-}emb\ o\ s) = mg\text{-}of\text{-}pz\ o\ unfold\text{-}pz\ s$
$\bigwedge s.\ unfold\text{-}mg\ (mg\text{-}of\text{-}alt1\text{-}emb\ o\ s) = mg\text{-}of\text{-}alt1\ o\ unfold\text{-}alt\ s$
$\bigwedge s.\ unfold\text{-}mg\ (mg\text{-}of\text{-}alt2\text{-}emb\ o\ s) = mg\text{-}of\text{-}alt2\ o\ unfold\text{-}alt\ s$
$\bigwedge s.\ unfold\text{-}pz\ (pz\text{-}of\text{-}alt1\text{-}emb\ o\ s) = pz\text{-}of\text{-}alt1\ o\ unfold\text{-}alt\ s$
$\bigwedge s.\ unfold\text{-}pz\ (pz\text{-}of\text{-}alt2\text{-}emb\ o\ s) = pz\text{-}of\text{-}alt2\ o\ unfold\text{-}alt\ s$
by *commute-prover+*

method *inj-prover* =
intro injI,
match **conclusion in** $x = y$ **for** $x\ y \Rightarrow$ ⟨*coinduction arbitrary*: *x y*, *fastforce*⟩

lemma *inj*:

inj lts-of-dlts
inj react-of-dlts
inj gen-of-lmc
inj lmdp-of-lmc
inj sseg-of-lmdp
inj sseg-of-react
inj sseg-of-lts
inj ssegopt-of-alt
inj seg-of-gen
inj seg-of-sseg
inj bun-of-lts
inj bunopt-of-alt
inj bun-of-gen
inj pz-of-seg
inj pz-of-bun
inj str-of-mc
inj alt-of-str
inj ssegopt-of-sseg
inj segopt-of-seg
inj bunopt-of-bun
inj pzopt-of-pz
inj pzopt-of-mg
inj mg-of-pzopt

inj mg-of-pz
inj mg-of-alt1
inj mg-of-alt2
inj pz-of-alt1
inj pz-of-alt2
by *inj-prover+*

end

lemma *hierarchy*:
$\bigwedge$*s1 s2. bisimilar-dlts s1 s2 x y* $\longleftrightarrow$ *bisimilar-lts* (*lts-of-dlts-emb o s1*) (*lts-of-dlts-emb o s2*) *x y*
$\bigwedge$*s1 s2. bisimilar-lmc s1 s2 x y* $\longleftrightarrow$ *bisimilar-gen* (*gen-of-lmc-emb o s1*) (*gen-of-lmc-emb o s2*) *x y*
$\bigwedge$*s1 s2. bisimilar-lmc s1 s2 x y* $\longleftrightarrow$ *bisimilar-lmdp* (*lmdp-of-lmc-emb o s1*) (*lmdp-of-lmc-emb o s2*) *x y*
$\bigwedge$*s1 s2. bisimilar-dlts s1 s2 x y* $\longleftrightarrow$ *bisimilar-react* (*react-of-dlts-emb o s1*) (*react-of-dlts-emb o s2*) *x y*
$\bigwedge$*s1 s2. bisimilar-lmdp s1 s2 x y* $\longleftrightarrow$ *bisimilar-sseg* (*sseg-of-lmdp-emb o s1*) (*sseg-of-lmdp-emb o s2*) *x y*
$\bigwedge$*s1 s2. bisimilar-lts s1 s2 x y* $\longleftrightarrow$ *bisimilar-sseg* (*sseg-of-lts-emb o s1*) (*sseg-of-lts-emb o s2*) *x y*
$\bigwedge$*s1 s2. bisimilar-alt s1 s2 x y* $\longleftrightarrow$ *bisimilar-sseg* (*ssegopt-of-alt-emb o s1*) (*ssegopt-of-alt-emb o s2*) *x y*
$\bigwedge$*s1 s2. bisimilar-react s1 s2 x y* $\longleftrightarrow$ *bisimilar-sseg* (*sseg-of-react-emb o s1*)

(*sseg-of-react-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-sseg s1 s2 x y* ⟷ *bisimilar-seg* (*seg-of-sseg-emb o s1*) (*seg-of-sseg-emb o s2*) *x y*
⋀(*s1* :: - ⇒ (*'a option* × - *pmf*) *set*[-]) *s2*.
bisimilar-sseg s1 s2 x y ⟷ *bisimilar-seg* (*seg-of-sseg-emb o s1*) (*seg-of-sseg-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-gen s1 s2 x y* ⟷ *bisimilar-seg* (*seg-of-gen-emb o s1*) (*seg-of-gen-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-lts s1 s2 x y* ⟷ *bisimilar-bun* (*bun-of-lts-emb o s1*) (*bun-of-lts-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-alt s1 s2 x y* ⟷ *bisimilar-bun* (*bunopt-of-alt-emb o s1*) (*bunopt-of-alt-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-gen s1 s2 x y* ⟷ *bisimilar-bun* (*bun-of-gen-emb o s1*) (*bun-of-gen-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-seg s1 s2 x y* ⟷ *bisimilar-pz* (*pz-of-seg-emb o s1*) (*pz-of-seg-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-bun s1 s2 x y* ⟷ *bisimilar-pz* (*pz-of-bun-emb o s1*) (*pz-of-bun-emb o s2*) *x y*
⋀(*s1* :: - ⇒ (*'a option* × -) *pmf set*[-]) *s2*.
bisimilar-seg s1 s2 x y ⟷ *bisimilar-pz* (*pz-of-seg-emb o s1*) (*pz-of-seg-emb o s2*) *x y*
⋀(*s1* :: - ⇒ ((*'a option* × -) *set*[-]) *pmf*) *s2*.
bisimilar-bun s1 s2 x y ⟷ *bisimilar-pz* (*pz-of-bun-emb o s1*) (*pz-of-bun-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-mc s1 s2 x y* ⟷ *bisimilar-str* (*str-of-mc-emb o s1*) (*str-of-mc-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-sseg s1 s2 x y* ⟷ *bisimilar-sseg* (*ssegopt-of-sseg-emb o s1*) (*ssegopt-of-sseg-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-seg s1 s2 x y* ⟷ *bisimilar-seg* (*segopt-of-seg-emb o s1*) (*segopt-of-seg-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-bun s1 s2 x y* ⟷ *bisimilar-bun* (*bunopt-of-bun-emb o s1*) (*bunopt-of-bun-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-pz s1 s2 x y* ⟷ *bisimilar-pz* (*pzopt-of-pz-emb o s1*) (*pzopt-of-pz-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-str s1 s2 x y* ⟷ *bisimilar-alt* (*alt-of-str-emb o s1*) (*alt-of-str-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-mg s1 s2 x y* ⟷ *bisimilar-pz* (*pzopt-of-mg-emb o s1*) (*pzopt-of-mg-emb o s2*) *x y*
unfolding *inj*[*THEN inj-eq*] *bisimilar-alt emb-commute o-apply* **by** (*rule refl*)+

An edge that would make the graph cyclic

lemma
⋀*s1 s2*. *bisimilar-pz s1 s2 x y* ⟷ *bisimilar-mg* (*mg-of-pz-emb o s1*) (*mg-of-pz-emb o s2*) *x y*
unfolding *inj*[*THEN inj-eq*] *bisimilar-alt emb-commute o-apply* **by** (*rule refl*)+

Some redundant (historic) transitive edges

lemma
⋀*s1 s2*. *bisimilar-pz s1 s2 x y* ⟷ *bisimilar-mg* (*mg-of-pzopt-emb o s1*) (*mg-of-pzopt-emb*

o s2) *x y*
⋀*s1 s2*. *bisimilar-alt s1 s2 x y* ⟷ *bisimilar-mg* (*mg-of-alt1-emb o s1*) (*mg-of-alt1-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-alt s1 s2 x y* ⟷ *bisimilar-mg* (*mg-of-alt2-emb o s1*) (*mg-of-alt2-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-alt s1 s2 x y* ⟷ *bisimilar-pz* (*pz-of-alt1-emb o s1*) (*pz-of-alt1-emb o s2*) *x y*
⋀*s1 s2*. *bisimilar-alt s1 s2 x y* ⟷ *bisimilar-pz* (*pz-of-alt2-emb o s1*) (*pz-of-alt2-emb o s2*) *x y*
unfolding *inj*[*THEN inj-eq*] *bisimilar-alt emb-commute o-apply* **by** (*rule refl*)+

7 Some special proofs

Two views on LTS

lemma ∃*f*::((*'a* × *'s*) *set* ⇒ *'a* ⇒ *'s set*). *bij f*
by (*fastforce simp*: *bij-def inj-on-def fun-eq-iff image-iff*
intro: *exI*[*of* - λ*S a*. {*s*. (*a*, *s*) ∈ *S*}] *exI*[*of* - {(*a*, *b*). *b* ∈ *f a*} **for** *f*])

lemma ∃*f*::((*'a* × *'s*) *set*[(*'a* × *'s*) *set*] ⇒ *'a* ⇒ *'s set*[*'s set*]). *bij f*
by (*auto simp*: *bij-def inj-on-def fun-eq-iff image-iff bset-eq-iff*
intro!: *exI*[*of* - λ*S a*. *bCollect* (λ*s*. *bmember* (*a*, *s*) *S*)]
exI[*of* - *bCollect* (λ(*a*, *b*). *bmember b* (*f a*)) **for** *f*])

mc is trivial

lemma *mc-unit*:
fixes *x y* :: *mc*
shows *x* = *y*
by (*coinduction arbitrary*: *x y*)
(*auto simp*: *pmf.in-rel map-fst-pair-pmf map-snd-pair-pmf intro*: *exI*[*of* - *pair-pmf x y* **for** *x y*])

lemma *bisimilar-mc s1 s2 x y*
unfolding *bisimilar-alt* **by** (*rule mc-unit*)

8 Printing the Hierarchy Graph

ML ‹
local

val trim = *filter* (*fn s* => *s* <> "" *andalso s* <> *set*);

fun str-of-T (*Type* (*c*, *Ts*)) =
"" *space-implode* "" (*trim* [*commas* (*trim* (*map str-of-T Ts*)), *Long-Name.base-name c*])
| *str-of-T* - = "";

fun get-edge thm = *thm*

```
  |> Thm.concl-of
  |> HOLogic.dest-Trueprop
  |> HOLogic.dest-eq |> fst
  |> dest-comb |> fst
  |> fastype-of
  |> dest-funT
  |> apply2 str-of-T;

val edges = map get-edge @{thms hierarchy[unfolded bisimilar-alt emb-commute
o-apply, THEN iffD1]};

val nodes = distinct (op =) (maps (fn (x, y) => [x, y]) edges);

val node-graph = map (fn s => ((s, Graph-Display.content-node s []), [] : string
list)) nodes;

val graph = fold (fn (x, y) => fn g =>
  AList.map-entry (fn (x, (y, -)) => x = y) x (cons y) g) edges node-graph

in

val - = Graph-Display.display-graph graph

end
```

9 Vardi Systems

```
context notes [[bnf-internals]]
begin
  datatype ('a, 'b, 'k) var0 = PMF ('a × 'b) pmf | BPS ('a × 'b) set['k]
end

inductive var-eq :: ('a, 'b, 'k) var0 ⇒ ('a, 'b, 'k) var0 ⇒ bool (infixl ∼ 65) where
  var-eq-reflp[intro]: x ∼ x
| [intro]: PMF (return-pmf (a, x)) ∼ BPS (bsingleton (a, x))
| [intro]: BPS (bsingleton (a, x)) ∼ PMF (return-pmf (a, x))

lemma var-eq-symp: x ∼ y ⟹ y ∼ x
  by (auto elim: var-eq.cases)

lemma var-eq-transp: x ∼ y ⟹ y ∼ z ⟹ x ∼ z
  by (auto elim!: var-eq.cases)

quotient-type ('a, 'b, 'k) var = ('a, 'b, 'k) var0 / (∼)
  by (auto intro: equivpI reflpI sympI transpI  var-eq-symp var-eq-transp)

lift-definition map-var :: ('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ ('a, 'b, 'k) var ⇒ ('c, 'd, 'k)
```

```
var
  is map-var0
  by (auto elim!: var-eq.cases simp: map-bset-bsingleton)

lift-definition set1-var :: ('a, 'b, 'k) var ⇒ 'a set
  is set1-var0
  by (auto elim!: var-eq.cases)

lift-definition set2-var :: ('a, 'b, 'k) var ⇒ 'b set
  is set2-var0
  by (auto elim!: var-eq.cases)

inductive rel-var :: ('a ⇒ 'c ⇒ bool) ⇒ ('b ⇒ 'd ⇒ bool) ⇒ ('a, 'b, 'k) var ⇒
('c, 'd, 'k) var ⇒ bool for R S where
  set1-var x ⊆ {(x, y). R x y} ⟹ set2-var x ⊆ {(x, y). S x y} ⟹
    rel-var R S (map-var fst fst x) (map-var snd snd x)

abbreviation (input) var0-of-gen-emb ≡ case-option (BPS bempty) PMF
abbreviation (input) var0-of-lts-emb ≡ BPS

lift-definition var-of-gen-emb :: ('a × 'b) pmf option ⇒ ('a, 'b, 'k) var is var0-of-gen-emb
.
lift-definition var-of-lts-emb :: ('a × 'b) set['k] ⇒ ('a, 'b, 'k) var is var0-of-lts-emb
.

abbreviation bisimilar-var ≡ bisimilar (λR. rel-var (=) R)

lemma map-var0-eq-BPS-iff[simp]:
  map-var0 f g z = BPS X ⟷ (∃ Y. z = BPS Y ∧ map-bset (map-prod f g) Y =
X)
  by (cases z) auto

lemma map-var0-eq-PMF-iff[simp]:
  map-var0 f g z = PMF p ⟷ (∃ q. z = PMF q ∧ map-pmf (map-prod f g) q =
p)
  by (cases z) auto

lemma bisimilar-lts s1 s2 x y ⟷ bisimilar-var (var-of-lts-emb o s1) (var-of-lts-emb
o s2) x y
  (is - ⟷ bisimilar-var (?emb1 o -) (?emb2 o -) - -)
unfolding bisimilar-def o-apply proof safe
  fix R assume R x y and
    bis: ∀ x y. R x y ⟶ rel-bset (rel-prod (=) R) (s1 x) (s2 y)
  from ‹R x y› show ∃ R. R x y ∧ (∀ x y. R x y ⟶ rel-var (=) R (?emb1 (s1 x))
(?emb2 (s2 y)))
  proof (intro exI[of - R], safe)
    fix x y
    assume R x y
    with bis have ∗: rel-bset (rel-prod (=) R) (s1 x) (s2 y) by blast
```

```
    then obtain z where
      set-bset z ⊆ {(x, y). rel-prod (=) R x y} map-bset fst z = s1 x map-bset snd z
= s2 y
      by (auto simp: bset.in-rel)
    then show rel-var (=) R (?emb1 (s1 x)) (?emb2 (s2 y))
      unfolding rel-var.simps by (transfer fixing: z)
        (force simp: bset.map-comp o-def split-beta prod-set-simps
        intro: exI[of - BPS (map-bset (λ((a,b),(c,d)). ((a,c),(b,d))) z)])
  qed
next
  fix R assume R x y and
    bis: ∀ x y. R x y ⟶ rel-var (=) R (?emb1 (s1 x)) (?emb2 (s2 y))
  from ‹R x y› show ∃ R. R x y ∧ (∀ x y. R x y ⟶ rel-bset (rel-prod (=) R) (s1
x) (s2 y))
  proof (intro exI[of - R], safe)
    fix x y
    assume R x y
    with bis have rel-var (=) R (?emb1 (s1 x)) (?emb2 (s2 y)) by blast
    then obtain z where ∗:
      set1-var z ⊆ {(x, y). x = y} set2-var z ⊆ {(x, y). R x y}
      ?emb1 (s1 x) = map-var fst fst z ?emb2 (s2 y) = map-var snd snd z
      by (auto simp: rel-var.simps)
    then show rel-bset (rel-prod (=) R) (s1 x) (s2 y)
      by (transfer fixing: s1 s2) (fastforce simp: bset.in-rel bset.map-comp o-def
map-pmf-eq-return-pmf-iff
          split-beta[abs-def] map-prod-def subset-eq split-beta prod-set-defs elim!:
var-eq.cases
        intro: exI[of - map-bset (λ((a,b),(c,d)). ((a,c),(b,d))) z for z]
          exI[of - bsingleton ((a,c),(b,d)) for a b c d])
  qed
qed

lemma bisimilar-gen s1 s2 x y ⟷ bisimilar-var (var-of-gen-emb o s1) (var-of-gen-emb
o s2) x y
  (is - ⟷ bisimilar-var (?emb1 o -) (?emb2 o -) - -)
unfolding bisimilar-def o-apply proof safe
  fix R assume R x y and
    bis: ∀ x y. R x y ⟶ rel-option (rel-pmf (rel-prod (=) R)) (s1 x) (s2 y)
  from ‹R x y› show ∃ R. R x y ∧ (∀ x y. R x y ⟶ rel-var (=) R (?emb1 (s1 x))
(?emb2 (s2 y)))
  proof (intro exI[of - R], safe)
    fix x y
    assume R x y
    with bis have ∗: rel-option (rel-pmf (rel-prod (=) R)) (s1 x) (s2 y) by blast
    then show rel-var (=) R (?emb1 (s1 x)) (?emb2 (s2 y))
    proof (cases s1 x s2 y rule: option.exhaust[case-product option.exhaust])
      case None-None then show ?thesis unfolding rel-var.simps
        by (transfer fixing: s1 s2) (auto simp: bempty.rep-eq intro!: exI[of - BPS
bempty])
```

```
    next
      case (Some-Some p q)
      with ∗ obtain z where
        set-pmf z ⊆ {(x, y). rel-prod (=) R x y} map-pmf fst z = p map-pmf snd z
= q
        by (auto simp: pmf.in-rel)
      with Some-Some show ?thesis unfolding rel-var.simps
          by (transfer fixing: s1 s2 z) (force simp: pmf.map-comp o-def split-beta
prod-set-simps
          intro: exI[of - PMF (map-pmf (λ((a,b),(c,d)). ((a,c),(b,d))) z)])
    qed auto
  qed
next
  fix R assume R x y and
    bis: ∀ x y. R x y ⟶ rel-var (=) R (?emb1 (s1 x)) (?emb2 (s2 y))
  from ‹R x y› show ∃ R. R x y ∧ (∀ x y. R x y ⟶
     rel-option (rel-pmf (rel-prod (=) R)) (s1 x) (s2 y))
  proof (intro exI[of - R], safe)
    fix x y
    assume R x y
    with bis have rel-var (=) R (?emb1 (s1 x)) (?emb2 (s2 y)) by blast
    then obtain z where ∗:
        set1-var z ⊆ {(x, y). x = y} set2-var z ⊆ {(x, y). R x y}
        ?emb1 (s1 x) = map-var fst fst z ?emb2 (s2 y) = map-var snd snd z
        by (auto simp: rel-var.simps)
    then show rel-option (rel-pmf (rel-prod (=) R)) (s1 x) (s2 y)
    proof (cases s1 x s2 y rule: option.exhaust[case-product option.exhaust])
      case Some-None
     with ∗ show ?thesis by transfer (auto simp: bempty.rep-eq elim!: var-eq.cases)
    next
      case None-Some
      with ∗ show ?thesis by transfer (auto elim!: var-eq.cases)
    next
      case (Some-Some p q)
      with ∗ show ?thesis
        by transfer (fastforce simp: subset-eq split-beta prod-set-defs
          elim!: var-eq.cases intro!: rel-pmf-reflI)
    qed simp
  qed
qed
```